Datentypen in c

1 Motivation

Bisher haben wir lediglich mit den Datentypen gearbeitet, die c von sich aus bereitstellt (int, long, float, double, etc). Dies sind die sog. eingebauten (built-in) oder primitiven Datentypen. In diesem Tutorium werden wir lernen, wie wir Datentypen in c selbst definieren. Des weiteren werden wir lernen, dass Arrays in c etwas anderes sind als in Java. Und wir werden sehen, dass c in gewisser Weise auch eine objektorientierte und sogar in Teilen eine funktionale Sprache ist. Viele Konzepte werdet ihr also bereits kennen. Allerdings ist c eine wesentlich hardwarenähere Sprache, die nicht von sich aus eine elegante Syntax zur Verfügung stellt, um objektorientiert und/oder funktional programmieren zu können. Daher werden wir sehr viel mehr ``unter der Motorhaube" arbeiten müssen, was c einerseits anstrengend und auch fehleranfällig macht, uns aber hoffentlich ein besseres Verständnis dafür beschert, wie diese abstrakten Konzepte eigentlich auf der Maschine implementiert werden.

2 Enums

Enumerationen oder kurz *enums* sind sog. Aufzählungstypen. So könnte man den Typ *Woche* als eine Aufzählung der Wochentage *Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag* und *Sonntag* betrachten. Ihr habt dieses Konzept übrigens schon kennengelernt, nämlich in Opal. Erinnert ihr euch noch, wie das dort hieß? Richtig, es handelt sich um Summentypen mit Konstanten als Varianten des Typs. Allerdings sind diese Varianten in c nichts anderes als benannte Ganzzahlen, und im Gegensatz zu Opal kann man auf sie auch alle Operationen anwenden, die auf Ganzzahlen erlaubt sind. Mittwoch+1 wäre also Donnerstag. Was man durch ihre Verwendung allerdings gewinnt, ist eine wesentlich bessere Lebarkeit des Code. Wir werden gleich ein Beispiel sehen.

2.1 Boolsche Werte

C kennt keinen eigenen Datentypen bool. Das **false** anderer Sprachen entspricht in c dem Integerwert 0, alle anderen Werte bedeuten **true**. Erst ab C99 gibt es die in der Datei ``stdbool.h" definierten Konstanten **true** und **false**. Wenn wir aber aus irgendwelchen Gründen kein c99 benutzen wollen oder dürfen, können wir uns den Datentypen **bool** mittels Enumerationen aber auf denkbar einfache Art und Weise selbst definieren. Nämlich als:

```
enum bool{
  FALSE,
  TRUE
};
```

Implizit entspricht nun false dem Wert 0 und true dem Wert 1.

Konstanten groß zu schreiben ist dabei in c übliche Praxis, um eine versehentliche Zuweisung zu vermeiden.

Statt

```
int flag= 1;
```

können wir nun schreiben

```
enum bool flag= TRUE;
```

Das **enum** vor dem bool sieht zugegeben etwas seltsam aus. Wir können aber auch dies umgehen, denn c stellt uns einen Mechanismus zur Verfügung, um Typen umzubennen, ihnen neue und lesbarere Namen zu geben, nämlich **typedef**. Ob man das möchte, sei im Übrigen jedem freigestellt, manche Programmierer schätzen an der obigen Schreibweise gerade die Tatsache, dass man explizit erkennt, dass es sich um einen Aufzählungstyp handelt. Wenn ihr euch aber zur ersteren Fraktion zählt, so könnt ihr das enum loswerden:

```
typedef enum bool{
  FALSE,
  TRUE
} boolean;
```

Jetzt können wir, wie in Java schreiben:

```
boolean flag= TRUE;
```

und sämtliche logischen Operatoren wie &&, ||, etc. auf Elemente unseres neuen Typs anwenden.

2.2 Übungsaufgabe: Farben

Standardmäßig numeriert c alle Varianten eines Aufzählungstyps konsekutiv von 0 an durch. Wir können jedoch auch explizit Werte vergeben. Als Beispiel möchten wir einen Typ **Color** definieren. Eine Farbe können wir als RGB(Rot-Grün-Blau)-Wert hexadezimal folgendermassen darstellen:

```
const int RED= 0xFF0000;
```

wobei jeweils zwei Ziffern für einen Farbkanal stehen. Jedoch fehlt hier die Typinformation, dass es sich um eine Farbe handelt, und nicht um ein beliebiges int.

Schreibe eine Datenstruktur **Color**, die die Grundfarben **black**, **white**, **red**, **yellow**, **green**, **cyan**, **blue**, und **magenta** definiert, indem den einzelnen Varianten explizit Werte zugewiesen werden!

Lösung:

```
typedef enum color{
  WHITE= 0xFFFFFF,
  BLACK= 0x000000,
  RED= 0xFF0000,
  YELLOW= 0xFFFF00,
```

```
  GREEN= 0x00FF00,
  CYAN= 0x00FFFF,
  BLUE= 0x0000FF,
  MAGENTA= 0xFF00FF
} Color;
```

Es ist übrigens im Allgemeinen nicht immer nötig, jeder der Varianten einen Wert zuzuweisen, es wird dann einfach fortlaufend vom letzten zugewiesenen Wert hochgezählt, so lange, bis eine neue Zuweisung erfolgt.

3 Arrays

Die Definition eines (statischen - dass es noch andere Arten von Arrays gibt, werden wir in Vorlesung 2 kennenlernen) Arrays sieht erst einmal genauso aus wie in Java. Als Beispiel möchten wir einen Regenbogen programmieren.

```
Color rainbow[]= {RED, YELLOW, GREEN, CYAN, BLUE, MAGENTA};
```

Mit ncurses können wir den Regenbogen sogar farbig auf dem Terminal darstellen. Allerdings müssen wir dazu die von ncurses vorgegebenen Datentypen verwenden. Das gesamte Programm **rainbow1.c** zum Ausprobieren findet ihr in den Vorgaben zum Tutorium:

Das Programm benötigt die Bibliothek *ncursesw* und wird mit dem Aufruf

```
gcc -std=c99 -Wall -o rainbow rainbow.c -lncursesw
```

kompiliert. Wenn ihr auf eurem System kein ncursesw installieren könnt, müsst ihr euch wohl oder übel damit begnügen, Hex-Werte auf der Konsole auszugeben.

Ihr müsst dabei nicht den gesamten Code verstehen. Eine Zeile fällt jedoch ins Auge:

```
 for(int i= 0; i<sizeof(rainbow)/sizeof(int); i++)
```

Die Funktion **sizeof** ist so etwas wie das **.length** der Java-Welt. Angewandt auf ein Array liefert sie die Länge dieses Arrays, allerdings in Bytes. Deswegen muss noch durch die Anzahl der Bytes geteilt werden, die ein Integer belegt. Geht allerdings niemals davon aus, dass das genau vier sein müssen! Anders als der Java-Standard macht der c-Standard keinerlei Aussagen darüber, wie breit ein Datentyp sein muss. Das ist von Platform zu Platform und von Compiler zu Compiler durchaus unterschiedlich. Daher benutzen wir zum Ermitteln der Wortbreite - ebenfalls **sizeof**. Es gibt also zwei **sizeof**: Eines, dass ein *Datum* als Argument erwartet, und eines, das einen *Typen* erwartet. Selbstverständlich funktioniert letzteres ebensogut mit selbstdefinierten Typen wie **Color**.

Soweit, so gut. Bisher sieht, abgesehen von Details, alles so aus wie in Java. Ist es aber nicht. Um das zu verstehen, müssen wir betrachten, wie ein Array im Speicher aussieht. Beginnen wir mit Java. In Java ist ein Array nichts anderes als ein Objekt. Wie von jedem anderen Objekt können wir zur Laufzeit dessen Adresse im Speicher abfragen, wir können einer Variable

des Typs **int[]** ein anderes Integer-Array zuweisen, und ein Array besitzt sogar eine Membervariable **length**, die uns jederzeit über die Länge des Arrays in Kenntnis setzt. Noch besser - das Java Laufzeitsystem verhindert sogar, dass wir versehentlich über die Grenze des Arrays hinausschiessen. Das alles geht in c nicht. Wieso? Weil es in c gar keine Arrays gibt - zumindest nicht zur Laufzeit. Was aber ist dann ein Array in c? Malen wir uns als Beispiel einfach unser Regenbogenbeispiel auf, mit fiktiven Adressen.

Die Abbildung zeigt, dass unser Array nichts anderes ist, als eine Folge von Integers, die hintereinanderweg im Speicher liegen (Wir nehmen in unserem Beispiel an, dass ein Integer auf unserer Platform vier Bytes lang ist). Für c ist ein Array nichts anderes als - *die Adresse des ersten Elements*, in unserem Falle also? Richtig - die Adresse 0x34FF5E08. Diese Adresse setzt der Compiler beim Übersetzen des Programms überall dort ein, wo er auf das Symbol **rainbow** trifft. Dass da noch weitere Integers dranhängen, hat ein c-Programm zur Laufzeit schlichtwegs bereits vergessen.

Eine kleine Anekdote noch nebenbei: Der Operator **[]** ist nichts anderes als eine andere Schreibweise für **+**. Er addiert einen Offset auf die Adresse 0x34FF5E08 (in unserem fiktiven Beispiel). Und da die Addition kommutativ ist, kann ich in der for-Schleife genauso schreiben: **i[rainbow]**. Probiert's aus!

Wieso funktioniert aber dann **sizeof**? Ganz einfach: Der Compiler baut sich zum Übersetzen eine Tabelle, die sog. *Symbol Table*, in der alle Symbole, zusammen mit ihren Adressen im Speicher, verzeichnet sind. Im Falle eines Arrays steht dort auch dessen Länge eingetragen, die der Compiler deswegen kennt, weil ihr sie ja bei der Deklaration des Arrays angegeben habt. Und wenn der Compiler auf ein **sizeof(rainbow)** stößt, guckt er einfach unter **rainbow** in seiner Symbol Table nach, und setzt den Wert ein, der da steht. **sizeof** ist also keine Funktion, sondern ein Compile-Time Operator. Der Unterschied zwischen c und Java ist einfach der, dass der c-Compiler nach dem Übersetzen die Symbol Table wegwirft. Das ist einer der Gründe, warum c wesentlich schneller, dafür aber auch unsicherer ist als Java, denn die Java-Virtual-Machine verwaltet zur Laufzeit noch ganz viele Informationen darüber, wie der Speicher strukturiert ist, plus Informationen über Namen, Typen, und Längen. All das muss der Programmierer in c per Hand machen.

Zurück zu **sizeof**. Sizeof funktioniert nur genauso lange, wie ich mein Array nicht einer anderen Variable zuweise. Anders ausgedrückt, es geht nur so lange gut, so lange das Array nicht den Namen wechselt, unter dem der Compiler in der Symbol Table nachgucken kann. Genau das passiert aber, wenn ich mein Array an eine Funktion übergebe. Bei einem Funktionsaufruf wird das Array (und was ist dieses Array? - Richtig - die Adresse seines ersten Elementes) in den Speicher der Funktion kopiert, oder anders ausgedrückt, einem Funktionsargument zugewiesen. Damit spreche ich das Array nun mit dem Namen des Funktionsarguments an - und der Compiler

kann damit in der Symbol Table keine zugehörige Länge mehr finden. Macht ja auch Sinn so, denn wer hindert mich denn daran, der Funktion Arrays ganz unterschiedlicher Länge übergeben zu können? Geht in Java ja auch. Sizeof funktioniert also nicht bei Funktionsargumenten. Fällt euch ein Ausweg ein? Genau, man muss die Länge des Arrays als *zusätzliches* Argument mitübergeben.

3.1 Übungsaufgabe: Arrays als Argumente von Funktionen

Am besten, wir betrachten ein Codebeispiel. Und zwar möchten wir in unserem obigen Codebeispiel die Zeilen

```
for(int i= 0; i<sizeof(rainbow)/sizeof(int); i++){
  attron(COLOR_PAIR(rainbow[i]));
  addstr("\u2588");
  attroff(COLOR_PAIR(rainbow[i]));
}
```

durch den Aufruf einer Funktion mit der Signatur

```
void printRainbow(int rainbow[], int rainbowLength);
```

ersetzen.

Lösung siehe **rainbow2.c**.

4 Structs

Das Codebeispiel aus der vorigen Aufgabe funktioniert zwar, mir persönlich gefällt die Lösung aber nicht sonderlich, insbesondere, wenn man sich vorstellt, ganz viele Arrays als Argumente übergeben zu wollen. Da ist die objektorientierte Lösung in Java doch schöner. Glücklicherweise hat aber auch c - wenn auch sehr rudimentär - so etwas wie Klassen, mit denen wir uns Java-Arrays einfach nachbauen können - nämlich **structs**. Ein Int-Array, das seine eigene Länge kennt, können wir als neue Klasse **Rainbow** ganz einfach folgendermassen definieren. Und jetzt ist auch klar, dass es sich nicht um irgendein Int-Array handelt, sondern um einen Regenbogen mit genau 6 Farben:

```
typedef struct rainbow{
  const int length;
  int color[6];
} Rainbow;
```

Sieht einer enum-Deklaration verdächtig ähnlich, oder? Zurück zu Opal: Welcher Art von Datentypen entspricht ein **struct**, wenn ein **enum** ein Summentyp ist? Richtig - einem Produkttypen.

Wichtig ist noch, dass wir, wenn wir ein statisches Array zur Compilezeit deklarieren, ohne es sofort zu initialisieren, auch gleich seine Länge in eckigen Klammern angeben. Das ist an

dieser Stelle zwar nicht schön, vor allem, weil man die Länge beim Initialisieren noch einmal explizit wiederholen muss, aber im Sinne der Wartbarkeit und des Information Hiding immer noch besser, als Konstanten weit verstreut im Code zu haben. Denn mit unserer Herangehensweise muss lediglich der Programmierer der Bibliothek die Länge kennen, Anwendercode ist dagegen völlig davon losgelöst.

Denn nun können wir, wie in Java mit **.length** auf die Länge einer Variable vom Typ Regenbogen und mit **.color** auf seine Farben zugreifen. Noch besser, c stellt uns sogar eine Art ``Konstruktor'' zur Verfügung, mit Hilfe dessen wir einen Regenbogen initialisieren können:

```
Rainbow rainbow={
  .length= 6,
 .color= {COLOR_RED, COLOR_YELLOW, COLOR_GREEN, COLOR_CYAN, COLOR_BLUE, COLOR_MAGENTA}
};
```

4.1 Übungsaufgabe: Der Regenbogen objektorientiert

Schreibe das Regenbogenprogramm so um, dass es unsere neue Klasse benutzt!

Lösung siehe **rainbow3.c**.

4.2 Übungsaufgabe: Colors Revisited

Farben als enums zu definieren, war zwar ein nettes Einsteigerbeispiel, bewährt sich in der Praxis aber nicht sonderlich, weil man die einzelnen Farbkanäle nicht gezielt verändern kann. Deklariert ein struct **color** mit den ``Membervariablen'' **red**, **green** uns **blue** und definiert unsere altbekannten Variablen **black**, **white**, **red**, **yellow**, **green**, **cyan**, **blue**, und **magenta** als ``Objekte''.

Lösung:

```
typedef struct color{
  unsigned char red;
  unsigned char green;
  unsigned char blue;
} Color;

const Color black={
  .red= 0x00,
  .green= 0x00,
  .blue= 0x00
};

const Color white={
  .red= 0xFF,
  .green= 0xFF,
  .blue= 0xFF
};
```

```
const Color red={
  .red= 0xFF,
  .green= 0x00,
  .blue= 0x00
};

const Color yellow={
  .red= 0xFF,
  .green= 0xFF,
  .blue= 0x00
};

const Color green={
  .red= 0x00,
  .green= 0xFF,
  .blue= 0x00
};

const Color cyan={
  .red= 0x00,
  .green= 0xFF,
  .blue= 0xFF
};

const Color blue={
  .red= 0x00,
  .green= 0x00,
  .blue= 0xFF
};

const Color magenta={
  .red= 0xFF,
  .green= 0x00,
  .blue= 0xFF
};
```

5 Unions

Eine letzte Art von Datentypen in c sind die sog. Unions. Eine Union deklariert man genauso wie ein Struct, nur mit dem Schlüsselwort **union**. Der Unterschied ist, dass alle Variablen, die wir darin deklarieren, nur andere Bezeichner für *ein- und dieselbe Speicherstelle* sind! Wir können also z.B. unser Farbstruct zum Modifizieren einer Farbe benutzen, zum Anzeigen aber bequem die Hexadezimaldarstellung benutzen, wenn wir nur beides in eine Union packen. Denn beide benötigen genau 3 Byte speicherplatz.

```
typedef union color{
```

```
  struct rgb{
    unsigned char red;
    unsigned char green;
    unsigned char blue;
  } rgb;
  int hex;
} Color;
```

Zu beachten ist allerdings, und das gilt generell für die hexadezimale Darstellung, dass die Sinnhaftigkeit der Farben von der Endianness der Maschine abhängt.

5.1 Zur Bearbeitung in der Übung empfohlen

iee754

6 Zusatzaufgabe: 2D-Arrays (Zur Bearbeitung in der anschliessenden Übung empfohlen)

Als nächstes möchten wir uns vornehmen, nicht nur einen Regenbogen zu malen, sondern den ganzen Bildschirm voll mit bunten Kästchen. Problem: Der Bildschirm ist zweidimensional. Also können wir ihn am besten mit einem zweidimensionalen Array abbilden. Das können wir beispielsweise folgendermaßen deklarieren:

```
int screen[4][3];
```

Unser kleiner Bildschirm (ihr könnt euren natürlich gerne größer machen) hat nun die Breite vier und die Höhe drei Pixel. Welche der Dimensionen unseres Arrays wir dabei als Breite bzw. Höhe interpretieren, ist dabei uns selbst überlassen. Wir müssen nur aufpassen, die verschachtelte **for**-Schleife entsprechend aufzubauen.

Als nächstes definieren wir uns eine Funktion, die uns eine zufällige Farbe generiert:

```
static int randomColor(Rainbow rainbow){
  return rainbow.color[random()%rainbow.length];
}
```

und initialisieren unseren Bildschirm mit lustigen bunten Zufallsfarben.

```
for(int y= 0; y<sizeof(screen[0])/sizeof(int); y++){
  for(int x= 0; x<(sizeof(screen)/sizeof(screen[0])); x++){
    screen[x][y]= randomColor(rainbow);
  }
}
```

Betrachtet genau die Abbruchbedingungen und überlegt euch, weshalb das so funktionieren muss.

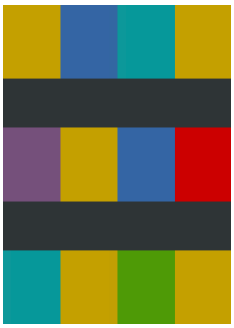
6.1 Programmierung der Ausgabefunktion

Zur Übung schreibt euch nun die zugehörige Ausgabe selbst!

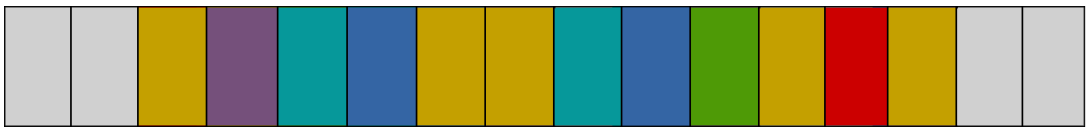
Lösung siehe **screen1.c**:

6.2 Speicherlayout mehrdimensionaler Arrays

Wie sieht nun aber das Ganze im Speicher aus? Denn der ist ja nun mal eindimensional. Angenommen, unser Programm produziert folgende Ausgabe:



Dann liegen die Spalten unseres Arrays hintereinanderweg im Speicher:



Die Indizierung über **[][]** wird praktischerweise zur Compiletime in einen eindimensionalen Index umgerechnet.

Das bedeutet insbesondere aber, dass bei einem mehrdimensionalen Array als Funktionsargument in allen ausser der linkesten eckigen Klammer die Längen *explizit* angegeben werden müssen! Der Compiler kann sonst keinen Code für die Indizierung generieren!

6.3 Mehrdimensionale Arrays als Funktionsargumente

Deklariere und definiere zwei Funktionen **initScreen** und **printScreen**, und benutze sie statt der for-Schleife, um deinen **screen** zu initialisieren und auszugeben!

Lösung siehe **screen2.c**:

6.4 Mehrdimensionale Arrays objektorientiert

Hartgecoded die Arraylängen im ganzen Programm zu verstreuen ist hässlich, hässlich, hässlich (und absolut unwartbar)! Ich überlasse es daher euch als Übung, auch den Code für 2D-Arrays

objektorientiert umzuschreiben.

7 Ausblick: Warum c eigentlich eine Multiparadigmensprache ist (Vorsicht - fortgeschrittener Stoff, nur für Hartgesottene)

RGB-Farben haben den Nachteil, dass man nicht ohne Kopfzerbrechen die Sättigung oder die Helligkeit einer gegebenen Farbe anpassen kann. Dafür gibt es ein Farbmodell, das wesentlich besser geeignet ist, nämlich HSB (Hue, Saturation, Brightness). Wir wollen unser Ziel, in Java objektorientiert zu programmieren, nun noch etwas weiter treiben, indem wir zwei Varianten (in Java würde man sagen, Subklassen) des Typs Color definieren.

7.1 Übungsaufgabe: Farbmodelle

Als erstes brauchen wir einen Typ für Farbmodelle. Welcher der Typen, die wir kennengelernt haben eignet sich dafür ganz hervorragend? Richtig, der Aufzählungstyp. Definiere also ein **enum** mit den beiden Varianten **RGB** und **HSB**!

Lösung:

```
typedef enum colorModel{
  RGB,
  HSB
} ColorModel;
```

7.2 Subtypen

Das Sprachmittel der Vererbung kennt c leider nicht, wohl aber die Variantenbildung. Wir müssen unser Ansinnen also wohl oder übel etwas ``von hinten durch die Brust ins Auge'' lösen. Zunächst stellen wir die Überlegung an, dass eine RGB-Farbe genausoviel Speicher belegt, wie eine HSB-Farbe. Wir können also einfach beide Varianten in ein- und demselben **struct** unterbringen. Und dafür brauchen wir **unions**.

In unserem speziellen Fall verzichten wir ausnahmsweise darauf, der Union einen Namen zu geben: Das nennt man dann *anonyme Union* und ist eine gcc-Erweiterung, die das Compilerflag **-fms-extensions** erfordert. Das Ganze sieht folgendermaßen aus (und ist absolut alles andere als cross-platform):

```
typedef struct color{
  ColorModel type;
  union{
    unsigned char red;
    unsigned char hue;
  };
  union{
    unsigned char green;
```

```
    unsigned char saturation;
  };
  union{
    unsigned char blue;
    unsigned char brightness;
  };
} Color;
```

Was bringt uns das jetzt nun? Vielleicht sehen wir einen Vorteil, wenn wir beginnen, Methoden auf unsere beiden Subklassen zu definieren. Java hat zum Beispiel die nützliche Methode toString(). Schreiben wir uns also eine Ausgabemethode für Farben:

```
void print(Color color){

  if(color.type == RGB)
    printf("R: %x G: %x B: %x", color.red, color.green, color.blue);
  else
    printf("H: %x S: %x B: %x", color.hue, color.saturation, color.brightness);
}
```

Diese Methode tut jetzt tatsächlich, abhängig vom Typen, etwas anderes - und das ist Objektorientierung! Ihr seid nicht überzeugt? Vielleicht werdet ihr sagen: Aber das sind doch keine Methoden! Methoden müssen doch Klassen zugeordnet sein!

Erstens ist das keineswegs in allen objektorientierten Sprachen so, das ist eher eine Erfindung der Java- und C++ Fraktion unter den Sprachentwicklern. Aber wenn ihr darauf besteht, eine Methode über den Punkt-Operator aufzurufen - auch das geht in c und benötigt nur eine kleine Modifikation unserer Color-Klasse.

```
typedef struct color{
  ColorModel type;
  union{
    unsigned char red;
    unsigned char hue;
  };
  union{
    unsigned char green;
    unsigned char saturation;
  };
  union{
    unsigned char blue;
    unsigned char brightness;
  };
  void (*print) (struct color c);
} Color;

void print(Color color){

  if(color.type == RGB)
```

```
    printf("R: %x G: %x B: %x", color.red, color.green, color.blue);
  else
    printf("H: %x S: %x B: %x", color.hue, color.saturation, color.brightness);
}
```

Die Farbe **rot** kann ich dann folgendermassen initialisieren (Die Membervariable und die Funktion, die ihr zuweise müssen übrigens nicht gleich heissen):

```
Color red={
  .type= RGB,
  {.red= 0xFF},
  {.green= 0x00},
  {.blue= 0x00}
  .print= print;
};
```

Und nun kann ich wie in Java schreiben:

```
red.print(red);
```

Das zusätzliche Argument der Methode kann ich allerdings nicht vermeiden. Da hat dann wie gesagt die c-Syntax ihre Grenzen. Was ich aber nun tun könnte, wäre verschiedenen Varianten meiner Klasse unterschiedliche Methoden zuzuordnen, indem ich der Membervariable **print** abhängig von der Membervariable **type** unterschiedliche Funktionen **printRGB** bzw **printHSB** zuweise.

Warum funktioniert das Ganze? Na, weil, wie ich eingangs schon behauptet habe, c eine funktionale Sprache ist - und das bedeutet, dass ich wie in Opal mit **let** Variablen Funktionen zuweisen kann. Eine Funktion kann man sich nämlich selbst wieder vorstellen als ein Objekt, das irgendwo im Speicher liegt.

Das heisst insbesondere, dass es in c auch Higher Order Functions gibt, d.h, dass ich Funktionen Funktionen als Argumente übergeben kann, und dass Funktionen Funktionen zurückliefern können! Wenn das auch zugegeben im Vergleich zu Opal nur sehr eingeschränkt funktioniert. Sowas Elegantes wie Currying gibt es hier selbstverständlich nicht. Als kleine Fingerübung könnt ihr ja mal versuchen, ausgehend von folgendem Codegerüst ein map auf **rainbows** zu implementieren und es benutzen, um beispielsweise die Sättigung oder die Helligkeit eures Regenbogens anzupassen! Ihr könnt der Funktion f natürlich auch weitere Argumente spendieren. Der Phantasie sind keine Grenzen gesetzt. Viel Spaß!

```
typedef struct rainbow{
  const int length;
  Color color[6];
  struct rainbow (*map) (struct rainbow colors, Color (*f) (Color color));
} Rainbow;
```